1. Pandas objects

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

If you have already used Python, you know about its standard data structures (list, dicts etc). If you use Python for science, you also probably know Numpy arrays which underlying almost all other specialized scientific packages.

None of these structures offers a simple way to handle database style data, nor to easily do standard database operations. This is why Pandas exists: it offers a complete ecosystem of structures and functions dedicated to handle large tables with inhomogeneous contents.

In this first chapter, we are going to learn about the two main structures of Pandsa: Series and Dataframes.

1.1 Series

1.1.1 Simple series

Series are a the Pandas version of 1-D Numpy arrays. To understand their specificities, let's create one. Usually Pandas structures (Series and Dataframes) are creates from other simpler structures like Numpy arrays or dictionaries:

(MZ: Pandas builds on numpy; Series are equivalent of a list)

In [2]:
```
numpy_array = np.array([4,8,38,1,6])
```

In [3]:
```
# MZ:
numpy_array**2
```

Out[3]:
```
array([  16,   64, 1444,    1,   36])
```

The function `pd.Series()` allows us to convert objects into Series:

In [4]:
```
pd_series = pd.Series(numpy_array)
pd_series
# on the left => the indices; can be anything, e.g. names of towns
```

Out[4]:
```
0     4
1     8
2    38
3     1
4     6
dtype: int64
```

The underlying structure can be recovered with the `.values` attribute:

In [5]:
```
pd_series.values
# MZ: output is numpy array
```

Out[5]:
```
array([ 4,  8, 38,  1,  6])
```

Otherwise, indexing works as for regular arrays:

In [6]:
```
pd_series[1]
```

Out[6]:
```
8
```

1.1.2 Indexing

On top of accessing values in a series by regular indexing, one can create custom indices for each element in the series:

In [7]:
```
pd_series2 = pd.Series(numpy_array, index=['a', 'b', 'c', 'd','e'])
# MZ: force the index to be what we give
# MZ: to retrieve the indexes (keys)
pd_series2.keys()
```

Out[7]:
```
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

In [8]:
```
pd_series2
```

Out[8]:
```
a     4
b     8
c    38
d     1
e     6
dtype: int64
```

Now a given element can be accessed either by using its regular index:

In [9]:
```
pd_series2[1]
```

Out[9]:
```
8
```

or its chosen index:

In [10]:
```
pd_series2['b']
```

Out[10]:
```
8
```

A more direct way to create specific indexes is to transform as dictionary into a Series:

In [11]:
```
# MZ: use dict to build the Series
composer_birth = {'Mahler': 1860, 'Beethoven': 1770, 'Puccini': 1858, 'S
hostakovich': 1906}
```

In [12]:
```
pd_composer_birth = pd.Series(composer_birth)
pd_composer_birth
```

Out[12]:
```
Beethoven       1770
Mahler          1860
Puccini         1858
Shostakovich    1906
dtype: int64
```

In [13]:
```
pd_composer_birth['Puccini']
```

Out[13]:
```
1858
```

1.2 Dataframes

In most cases, one has to deal with more than just one variable, e.g. one has the birth year and the death year of a list of composers. Also one might have different types of information, e.g. in addition to numerical variables (year) one might have string variables like the city of birth. The Pandas structure that allow one to deal with such complex data is called a Dataframe, which can somehow be seen as an aggregation of Series with a common index.

1.2.1 Creating a Dataframe

To see how to construct such a Dataframe, let's create some more information about composers:

In [14]:
```
composer_death = pd.Series({'Mahler': 1911, 'Beethoven': 1827, 'Puccini
': 1924, 'Shostakovich': 1975})
composer_city_birth = pd.Series({'Mahler': 'Kaliste', 'Beethoven': 'Bonn
', 'Puccini': 'Lucques', 'Shostakovich': 'Saint-Petersburg'})
```

Now we can combine multiple series into a Dataframe by precising a variable name for each series. Note that all our series need to have the same indices (here the composers' name):

In [15]:
```
# MZ: put Series together into a Dataframe
composers_df = pd.DataFrame({'birth': pd_composer_birth, 'death': compos
er_death, 'city': composer_city_birth})
composers_df
```

Out[15]:

	birth	city	death
Beethoven	1770	Bonn	1827
Mahler	1860	Kaliste	1911
Puccini	1858	Lucques	1924
Shostakovich	1906	Saint-Petersburg	1975

A more common way of creating a Dataframe is to construct it directly from a dictionary of lists:

In [16]:
```
dict_of_list = {'birth': [1860, 1770, 1858, 1906], 'death':[1911, 1827,
1924, 1975],
 'city':['Kaliste', 'Bonn', 'Lucques', 'Saint-Petersburg']}
```

In [17]:
```
pd.DataFrame(dict_of_list)
# MZ: default indexes from 0 to 3
```

Out[17]:

	birth	city	death
0	1860	Kaliste	1911
1	1770	Bonn	1827
2	1858	Lucques	1924
3	1906	Saint-Petersburg	1975

However we now lost the composers name. We can enforce it by providing, as we did before for the Series, a list of indices:

In [18]:
```
pd.DataFrame(dict_of_list, index=['Mahler', 'Beethoven', 'Puccini', 'Sho
stakovich'])
# MZ: you can explicitly pass the index
```

Out[18]:

	birth	city	death
Mahler	1860	Kaliste	1911
Beethoven	1770	Bonn	1827
Puccini	1858	Lucques	1924
Shostakovich	1906	Saint-Petersburg	1975

1.2.2 Accessing values

There are multiple ways of accessing values or series of values in a Dataframe. Unlike in Series, a simple bracket gives access to a column and not an index, for example:

In [19]:
```
composers_df['city']
```

Out[19]:
```
Beethoven                  Bonn
Mahler                  Kaliste
Puccini                 Lucques
Shostakovich    Saint-Petersburg
Name: city, dtype: object
```

returns a Series. Alternatively one can also use the *attributes* synthax and access columns by using:

In [20]:
```
composers_df.city
# rather recommended to use the brackets
```

Out[20]:
```
Beethoven                  Bonn
Mahler                  Kaliste
Puccini                 Lucques
Shostakovich    Saint-Petersburg
Name: city, dtype: object
```

The attributes synthax has some limitations, so in case something does not work as expected, revert to the brackets notation.

When specifiying multiple columns, a DataFrame is returned:

In [21]:
```
composers_df[['city', 'birth']]
```

Out[21]:

	city	birth
Beethoven	Bonn	1770
Mahler	Kaliste	1860
Puccini	Lucques	1858
Shostakovich	Saint-Petersburg	1906

One of the important differences with a regular Numpy array is that here, regular indexing doesn't work:

In [22]:
```
#composers_df[0,0]
```

Instead one has to use either the `.iloc[]` or the `.loc[]` method. `.ìloc[]` can be used to recover the regular indexing:

In [23]:
```
# MZ: recover by positions
composers_df.iloc[0,1]
```

Out[23]:
```
'Bonn'
```

While `.loc[]` allows one to recover elements by using the **explicit** index, on our case the composers name:

In [24]:
```
# MZ: recover by indices
composers_df.loc['Mahler','death']
```

Out[24]:
```
1911
```

Remember that `loc` and ``ìloc``` use brackets [] and not parenthesis ().

Numpy style indexing works here too

In [25]:
```
composers_df.iloc[1:3,:]
```

Out[25]:

	birth	city	death
Mahler	1860	Kaliste	1911
Puccini	1858	Lucques	1924

If you are working with a large table, it might be useful to sometimes have a list of all the columns. This is given by the `.keys()` attribute:

In [26]:
```
composers_df.keys()
```

Out[26]:
```
Index(['birth', 'city', 'death'], dtype='object')
```

1.2.3 Adding columns

It is very simple to add a column to a Dataframe. One can e.g. just create a column a give it a default value that we can change later:

In [27]:
```
# MZ: if pass a single value, add the same value everywhere
composers_df['country'] = 'default'
```

In [28]:
```
composers_df
```

Out[28]:

	birth	city	death	country
Beethoven	1770	Bonn	1827	default
Mahler	1860	Kaliste	1911	default
Puccini	1858	Lucques	1924	default
Shostakovich	1906	Saint-Petersburg	1975	default

Or one can use an existing list:

In [29]:
```
country = ['Austria','Germany','Italy','Russia']
```

In [30]:
```
composers_df['country2'] = country
```

In [31]:
```
composers_df
```

Out[31]:

	birth	city	death	country	country2
Beethoven	1770	Bonn	1827	default	Austria
Mahler	1860	Kaliste	1911	default	Germany
Puccini	1858	Lucques	1924	default	Italy
Shostakovich	1906	Saint-Petersburg	1975	default	Russia